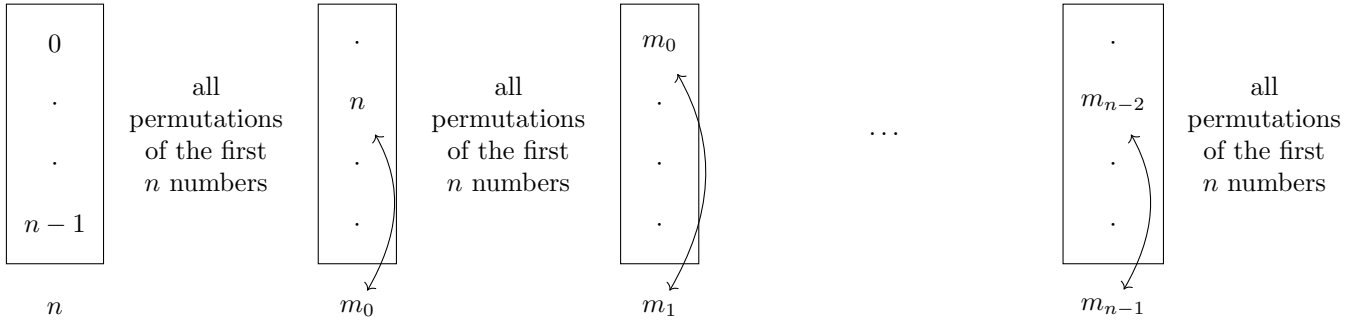


# NOTES ON CRYPTARITHM SOLVER AND PERMUTATIONS

ERIC MARTIN

## 1. HEAP'S ALGORITHM

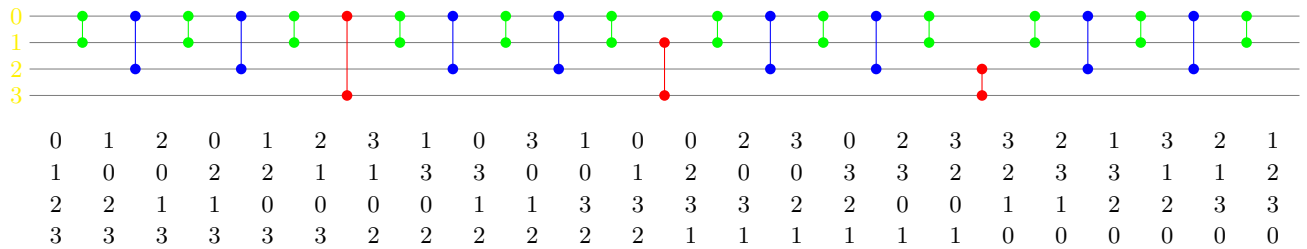
Let a nonzero natural number  $n$  be given. Heap's algorithm generates all permutations of a set  $S$  with  $n + 1$  elements, in such a way that any permutation, the first one excepted, is obtained from the previous one by exchanging two of  $S$ 's elements. Without loss of generality, take for  $S$  the set  $\{0, 1, \dots, n\}$ . The recursive version of Heap's algorithm can be illustrated as follows.



So the algorithm generates all permutations of the form  $L \star n$ , then all permutations of the form  $L \star m_0$ , then all permutations of the form  $L \star m_1$ , ..., and eventually all permutations of the form  $L \star m_{n-1}$ . The scheme is correct if  $\{m_0, m_1, \dots, m_{n-1}\} = \{0, \dots, n-1\}$ : at every stage, the algorithm has to select a new number from the first  $n$  ones (and exchange it with the current  $(n + 1)$ st number). Heap's algorithm uses the following strategy:

- In case  $n$  is odd, select the first number, then the second number, then the third number...
- In case  $n$  is even, always select the first number.

The following diagram illustrates with  $n = 3$ .



Note that starting with  $(0, 1, 2, 3)$ , Heap's algorithm produces  $(1, 2, 3, 0)$  as last permutation. We will see that starting with  $(0, 1, 2, 3, 4, 5)$ , it would produce  $(3, 4, 1, 2, 5, 0)$  as last permutation; starting with  $(0, 1, 2, 3, 4, 5, 6, 7)$ , it would produce  $(5, 6, 1, 2, 3, 4, 7, 0)$  as last permutation. More generally, starting with  $(0, 1, 2, \dots, 2n + 1)$ , Heap's algorithm will produce as last permutation  $(2n - 1, 2n, 1, 2, \dots, 2n - 2, 2n + 1, 0)$ .

Note that starting with  $(0, 1, 2)$ , Heap's algorithm produces  $(2, 1, 0)$  as last permutation. We will see that starting with  $(0, 1, 2, 3, 4)$ , it would produce  $(4, 1, 2, 3, 0)$  as last permutation; starting with  $(0, 1, 2, 3, 4, 5, 6)$ , it would produce  $(6, 1, 2, 3, 4, 5, 0)$  as last permutation. More generally, starting with  $(0, 1, 2, \dots, 2n)$ , Heap's algorithm will produce as last permutation  $(2n, 1, 2, \dots, 2n - 1, 0)$ .

The previous formulas can be used to generalise Heap's algorithm and generate all sequences of  $k$  numbers chosen from  $\{0, 1, \dots, n\}$ : when the last number has been selected, and the penultimate number has been selected, ..., and the

$(n - k + 1)$ st number has been selected, it suffices to stop the recursion and “simulate” all permutations of the remaining  $n + 1 - k$  numbers by applying those formulas.

The following is a possible recursive implementation of Heap’s algorithm.

```
def permute(L):
    yield from heap_permute(L, len(L))

def heap_permute(L, length):
    if length <= 1:
        yield L
    else:
        length -= 1
        for i in range(length):
            yield from heap_permute(L, length)
            if length % 2:
                L[i], L[length] = L[length], L[i]
            else:
                L[0], L[length] = L[length], L[0]
        yield from heap_permute(L, length)
```

The following is a possible iterative implementation of Heap’s algorithm.

```
def permute(L):
    yield L
    stack = [(0, i) for i in range(len(L) - 1, 0, -1)]
    while stack:
        low, high = stack.pop()
        if high % 2:
            L[low], L[high] = L[high], L[low]
        else:
            L[0], L[high] = L[high], L[0]
        yield L
        if low + 1 != high:
            stack.append((low + 1, high))
        for i in range(high - 1, 0, -1):
            stack.append((0, i))
```

## 2. PROOF OF CORRECTNESS

We prove that Heap’s algorithm is correct and that moreover, the following holds for all  $n \geq 1$ :

- (1) starting with  $(0, 1, 2, \dots, 2n)$ , all permutations of  $(0, 1, 2, \dots, 2n)$  are generated, ending in

$$(2n, 1, 2, \dots, 2n - 1, 0)$$

- (2) starting with  $(0, 1, 2, \dots, 2n + 1)$ , all permutations of  $(0, 1, 2, \dots, 2n + 1)$  are generated, ending in

$$(2n - 1, 2n, 1, 2, \dots, 2n - 2, 2n + 1, 0)$$

Proof is by induction. The base case  $n = 1$  is straightforward, so let  $n \geq 1$  be given, and assume that (1) holds. We show that (2) holds too.

- Starting from  $(0, 1, 2, \dots, 2n - 1, 2n) \star 2n + 1$ , Heap’s algorithm generates all permutations of the form  $L \star 2n + 1$ , ending with  $(2n, 1, 2, \dots, 2n - 1, 0) \star 2n + 1$ .
- Permuting first and last elements,  $(2n, 1, 2, \dots, 2n - 1, 0) \star 2n + 1$  is changed to  $(2n + 1, 1, 2, \dots, 2n - 1, 0) \star 2n$ . Starting from  $(2n + 1, 1, 2, \dots, 2n - 1, 0) \star 2n$ , the algorithm then generates all permutations of the form  $L \star 2n$ , ending with  $(0, 1, 2, \dots, 2n - 1, 2n + 1) \star 2n$ .

The first version is a minor adaptation of code written by Raymond Hettinger as part of [ActiveState Code Recipes](#). It uses the `permutations()` function, imported from `itertools` (and also the `findall()` function, imported from `re`). The second version does not import anything.

The first version filters out the permutations that assign 0 to one of the letters that start a word; the second version does not produce those permutations. The first version creates a string where the letters starting a word come first, followed by the letters not starting a word; it is the other way around for the second version. For instance, with the cryptarithm  $\text{SEND} + \text{MORE} == \text{MONEY}$ , the first version creates a string which could be  $\text{SMENDORY}$ , whereas the second version creates a string which could be  $\text{ENDORYSM}$ . Let us still use that example to explain how we proceed in the second version.

- Starting with the list  $L = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ , we use the generalisation of Heap's algorithm to generate lists of the form  $L_1L_2$ , one for each possible list  $L_2$  of two nonzero digits (so we ignore 0, as if  $L$  started with the element of index 1). This determines a possible assignment to  $\text{SM}$ .
- For each list of the form  $L_1L_2$  generated as described, we make a copy of  $L_1L_2$  and we use again the generalisation of Heap's algorithm to generate from the copy lists of the form  $L_{11}L_{12}L_2$ , one for each possible list  $L_{12}$  of six digits, amongst those in  $L_1$  (so 0 is now allowed but we ignore  $L_2$ , as if we were working with a list that ended at index 7). We return the last 8 digits of  $L_{11}L_{12}L_2$ , that is,  $L_{12}L_2$ , allocating the digits in  $L_{12}$  to  $\text{ENDORY}$  and the digits in  $L_2$  to  $\text{SM}$ .